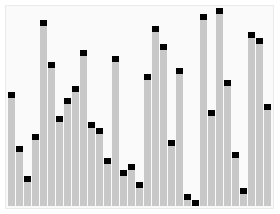
Gyorsrendezés

Ez a lap egy ellenőrzött változatarészletek megjelenítése/elrejtése

 [[bevezető szerkesztése](http://hu.wikipedia.org/w/index.php?title=Gyorsrendez%C3%A9s&action=edit&section=0&editintro=MediaWiki:Editintro-section-0)]

A Wikipédiából, a szabad enciklopédiából

[](http://hu.wikipedia.org/wiki/F%C3%A1jl:Sorting_quicksort_anim.gif)

Oszlopok magasság szerinti gyorsrendezése. A pirossal jelölt elem a támpont.

A **gyorsrendezés** vagy **quicksort** [algoritmus](http://hu.wikipedia.org/wiki/Algoritmus) egy [tömb](http://hu.wikipedia.org/wiki/T%C3%B6mb_(adatszerkezet)) elemeinek [sorba rendezésére](http://hu.wikipedia.org/wiki/Rendez%C3%A9s_(programoz%C3%A1s)). [C. A. R. Hoare](http://hu.wikipedia.org/w/index.php?title=C._A._R._Hoare&action=edit&redlink=1" \o "C. A. R. Hoare (a lap nem létezik)) találmánya, egyike azon rendezéseknek, amiknek a [bonyolultsága](http://hu.wikipedia.org/w/index.php?title=Bonyolults%C3%A1g&action=edit&redlink=1) átlagos esetben \Theta(n\log n). A gyorsrendezés általában gyorsabb az egyéb \Theta(n\log n) rendezéseknél, mert a belső ciklusa a legtöbb architektúrán nagyon hatékonyan implementálható, és az adatok jellegének ismeretében az algoritmus egyes elemei megválaszthatóak úgy, hogy csak nagyon ritkán fusson négyzetes ideig.

A gyorsrendezés egy [összehasonlító rendezés](http://hu.wikipedia.org/w/index.php?title=%C3%96sszehasonl%C3%ADt%C3%B3_rendez%C3%A9s&action=edit&redlink=1), és – hatékonyan implementálva – nem [stabil rendezés](http://hu.wikipedia.org/w/index.php?title=Stabil_rendez%C3%A9s&action=edit&redlink=1).

|  |
| --- |
| **Tartalomjegyzék**    [[elrejtés](http://hu.wikipedia.org/wiki/Gyorsrendez%C3%A9s)]   * [1 Története](http://hu.wikipedia.org/wiki/Gyorsrendez%C3%A9s#T.C3.B6rt.C3.A9nete) * [2 Működése](http://hu.wikipedia.org/wiki/Gyorsrendez%C3%A9s#M.C5.B1k.C3.B6d.C3.A9se)   + [2.1 Helyben rendező változat](http://hu.wikipedia.org/wiki/Gyorsrendez%C3%A9s#Helyben_rendez.C5.91_v.C3.A1ltozat) * [3 Bonyolultsága](http://hu.wikipedia.org/wiki/Gyorsrendez%C3%A9s#Bonyolults.C3.A1ga) * [4 Gyors kiválasztás](http://hu.wikipedia.org/wiki/Gyorsrendez%C3%A9s#Gyors_kiv.C3.A1laszt.C3.A1s) * [5 Kapcsolat más rendezőalgoritmusokkal](http://hu.wikipedia.org/wiki/Gyorsrendez%C3%A9s#Kapcsolat_m.C3.A1s_rendez.C5.91algoritmusokkal) * [6 Irodalom](http://hu.wikipedia.org/wiki/Gyorsrendez%C3%A9s#Irodalom) * [7 Hivatkozások](http://hu.wikipedia.org/wiki/Gyorsrendez%C3%A9s#Hivatkoz.C3.A1sok) * [8 További információk](http://hu.wikipedia.org/wiki/Gyorsrendez%C3%A9s#Tov.C3.A1bbi_inform.C3.A1ci.C3.B3k) * [9 Lásd még](http://hu.wikipedia.org/wiki/Gyorsrendez%C3%A9s#L.C3.A1sd_m.C3.A9g) |

Története [[szerkesztés](http://hu.wikipedia.org/w/index.php?title=Gyorsrendez%C3%A9s&action=edit&section=1)]

A gyorsrendezést 1960-ban, az [Elliott Brothers](http://hu.wikipedia.org/w/index.php?title=Elliott_Brothers&action=edit&redlink=1) angol számítógépgyártónak dolgozva fejlesztette ki Hoare.[[1]](http://hu.wikipedia.org/wiki/Gyorsrendez%C3%A9s" \l "cite_note-1)

Működése [[szerkesztés](http://hu.wikipedia.org/w/index.php?title=Gyorsrendez%C3%A9s&action=edit&section=2)]

A gyorsrendezés [oszd meg és uralkodj](http://hu.wikipedia.org/w/index.php?title=Oszd_meg_%C3%A9s_uralkodj&action=edit&redlink=1) elven működik: a rendezendő számok listáját két részre bontja, majd ezeket a részeket [rekurzívan](http://hu.wikipedia.org/w/index.php?title=Rekurz%C3%ADv&action=edit&redlink=1), gyorsrendezéssel rendezi. A felbontáshoz kiválaszt egy támpontnak nevezett elemet (más néven pivot, főelem vagy vezérelem), és particionálja a listát: a támpontnál kisebb elemeket eléje, a nagyobbakat mögéje mozgatja. [Teljes indukcióval](http://hu.wikipedia.org/wiki/Teljes_indukci%C3%B3" \o "Teljes indukció) könnyen belátható, hogy ez az algoritmus helyesen működik.

Az algoritmus pszeudokódja:

**function** quicksort(array)

**var** *list* less, equal, greater

**if** length(array) ≤ 1

**return** array

select a pivot value *pivot* from array

**for each** x **in** array

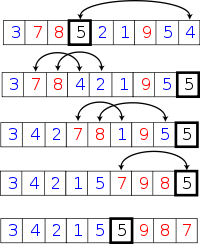
**if** x < pivot **then** append x to less

**if** x = pivot **then** append x to equal

**if** x > pivot **then** append x to greater

**return** concatenate(quicksort(less), equal, quicksort(greater))

**Helyben rendező változat**[[szerkesztés](http://hu.wikipedia.org/w/index.php?title=Gyorsrendez%C3%A9s&action=edit&section=3)]

[](http://hu.wikipedia.org/wiki/F%C3%A1jl:Partition_example.svg)

[http://bits.wikimedia.org/static-1.22wmf1/skins/common/images/magnify-clip.png](http://hu.wikipedia.org/wiki/F%C3%A1jl:Partition_example.svg)

Egy rövid lista helybeni rendezése. A keretes elem a támpont, a kék elemek nála kisebbek vagy egyenlőek, a pirosak nagyobbak vagy egyenlőek.

A fenti változat \Omega(n) tárhelyet igényel, azaz annyit, amennyit az [összefésüléses rendezés](http://hu.wikipedia.org/w/index.php?title=%C3%96sszef%C3%A9s%C3%BCl%C3%A9ses_rendez%C3%A9s&action=edit&redlink=1). A gyorsrendezés helyben is elvégezhető: az alábbi implementációban a *partition* eljárás az *array* tömb *left* és *right* közötti elemeit a *pivotIndex* elem két oldalára gyűjti:

**function** partition(array, left, right, pivotIndex)

pivotValue := array[pivotIndex]

swap array[pivotIndex] and array[right] *// Move pivot to end*

storeIndex := left

**for** i  **from**  left **to** right *// left ≤ i < right*

**if** array[i] ≤ pivotValue

swap array[i] and array[storeIndex]

storeIndex := storeIndex + 1

swap array[storeIndex] and array[right] *// Move pivot to its final place*

**return** storeIndex

**function** quicksort(array, left, right)

**if** right > left

select a pivot index (e.g. pivotIndex := left)

pivotNewIndex := partition(array, left, right, pivotIndex)

quicksort(array, left, pivotNewIndex-1)

quicksort(array, pivotNewIndex+1, right)

A particionálás során a sorrend megváltozik, azaz ez már nem stabil rendezés.

Bonyolultsága [[szerkesztés](http://hu.wikipedia.org/w/index.php?title=Gyorsrendez%C3%A9s&action=edit&section=4)]

Ha minden particionálás felezné a listát, akkor a rekurziót egy \log n mélységű fával lehetne ábrázolni, és mivel a fa minden szintjén összesen n elem van (csak egyre több partcícióban), az egyes szinteken a particionálásokhoz szükséges összes lépésszám \Theta(n) lenne, azaz az algoritmus összesen \Theta(n \log n) lépést igényelne. Ez általában is igaz, ha a nagyobb és a kisebb partíció aránya korlátos: ha például a kisebb partícióba mindig legalább az elemek 1%-a kerül, a fa még mindig legfeljebb 100 \log n mély. A legrosszabb esetben azonban, ha az egyik lista mindig egyelemű, a fa lineáris lesz, és n mélységű, azaz összesen \sum_{i=0}^n \Theta(n-i) = \Theta(n^2) lépés kell, vagyis a gyorsrendezés nem teljesít jobban, mint a [beszúrásos](http://hu.wikipedia.org/wiki/Besz%C3%BAr%C3%A1sos_rendez%C3%A9s) vagy a [kiválasztásos rendezés](http://hu.wikipedia.org/w/index.php?title=Kiv%C3%A1laszt%C3%A1sos_rendez%C3%A9s&action=edit&redlink=1).

Ha a támpontot véletlenszerűen választjuk, akkor a várható bonyoultság mindig \Theta(n \log n) lesz: minden lépésben 50% eséllyel választunk olyan támpontot, hogy a rövidebb listába legalább az elemek negyede kerül, és legfeljebb 2 \log_2 n ilyen vágás után eljutunk az egyelemű listáig, azaz a várható mélység 4 \log_2 n. Az átlagos eset lényegében azonos ezzel: az elemek egy véletlen permutációján futtatni az algoritmust ugyanaz, mintha véletlenül választanánk.

Formálisabban, az összehasonlítások átlagos száma a következő [rekurzív egyenlettel](http://hu.wikipedia.org/w/index.php?title=Rekurz%C3%ADv_egyenlet&action=edit&redlink=1) számítható:

C(n) = n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} (C(i)+C(n-i-1)) = 2n \ln n = 1.39n \log_2 n.

ahol n-1 a támpont összehasonlítása a partíció összes tőle különböző elemével, az összeg másik tagja pedig a lehetséges particionálások átlaga. Ez azt jelenti, hogy a gyorsrendezés átlagosan csak körülbelül 39%-kal lassabb, mint legjobb esetben.

A levezetés részletesebben: 
\begin{align}
 C(n) &= (n-1) + C \cdot \frac{n}{2} + C \cdot \frac{n}{2}\\
      &= (n-1) + 2C \cdot \frac{n}{2}\\
      &= (n-1) + 2\left(\frac{n}{2} - 1 + 2C \cdot \frac{n}{4} \right)\\
      &= n + n + 4C \cdot \frac{n}{4} - 1 - 2\\
      &= n + n + n + 8C \cdot \frac{n}{8} - 1 - 2 - 4\\
      &= \cdots\\
      &= kn + 2^kC \cdot \frac{n}{2^k} - (1 + 2 + 4 + \cdots + 2^{k-1}), \mbox{ where } \log_2 n > k > 0\\
      &= kn + 2^kC \cdot \frac{n}{2^k} - 2^k + 1,
      \rightarrow n \log_2 n + nC(1) - n + 1.
\end{align}


Gyors kiválasztás [[szerkesztés](http://hu.wikipedia.org/w/index.php?title=Gyorsrendez%C3%A9s&action=edit&section=5)]

A gyorsrendezés alapötlete [kiválasztó algoritmusoknál](http://hu.wikipedia.org/w/index.php?title=Kiv%C3%A1laszt%C3%B3_algoritmus&action=edit&redlink=1) is alkalmazható: ha egy lista *k*-adik legkisebb elemét kell kiválasztani, akkor a partíciók hosszából minden lépésben meg tudjuk mondani, melyikben van, tehát elég egyszeres rekurziót használni, amiből \Theta(n) átlagos futásidő adódik. Az algoritmus módosításával a legrosszabb esetbeni \Theta(n) idő is elérhető.

A \Theta(n) lépésigényű kiválasztó algoritmust a gyorsrendezésben felhasználva választhatjuk minden lépésben a [mediánt](http://hu.wikipedia.org/wiki/Medi%C3%A1n) támpontnak, így a futásidő a legrosszabb esetben is \Theta(n \log n) lesz. A gyakorlatban ezt ritkán használják, mert átlagosan valamivel lassabb.

Kapcsolat más rendezőalgoritmusokkal [[szerkesztés](http://hu.wikipedia.org/w/index.php?title=Gyorsrendez%C3%A9s&action=edit&section=6)]

A gyorsrendezés a [rendezőfa](http://hu.wikipedia.org/w/index.php?title=Rendez%C5%91fa&action=edit&redlink=1) egy speciális változatának is felfogható: ahelyett, hogy sorban beszúrnánk az elemeket egy fába, a rekurzív hívások adják a fastruktúrát.

A gyorsrendezés két fő vetélytársa a [kupacrendezés](http://hu.wikipedia.org/wiki/Kupacrendez%C3%A9s) és az [összefésüléses rendezés](http://hu.wikipedia.org/w/index.php?title=%C3%96sszef%C3%A9s%C3%BCl%C3%A9ses_rendez%C3%A9s&action=edit&redlink=1). Mindkettő átlagos és legrosszabb esetben is \Theta(n \log n) lépésigényű. Az összefésüléses rendezés stabil, és nagyon hatékony [láncolt listákon](http://hu.wikipedia.org/wiki/L%C3%A1ncolt_lista) és lassú elérésű tárban (például a [merevlemezen](http://hu.wikipedia.org/wiki/Merevlemez)) tárolt listákon, de tömbökön nagy a helyigénye. A kupacrendezésnek kisebb a helyigénye, mint a gyorsrendezésnek, de átlagban valamivel lassabb.

Irodalom [[szerkesztés](http://hu.wikipedia.org/w/index.php?title=Gyorsrendez%C3%A9s&action=edit&section=7)]

* Brian C. Dean, "A Simple Expected Running Time Analysis for Randomized 'Divide and Conquer' Algorithms." Discrete Applied Mathematics 154(1): 1-5. 2006.
* Hoare, C. A. R. "Partition: Algorithm 63," "Quicksort: Algorithm 64," and "Find: Algorithm 65." [Comm. ACM](http://hu.wikipedia.org/w/index.php?title=Comm._ACM&action=edit&redlink=1" \o "Comm. ACM (a lap nem létezik)) 4(7), 321-322, 1961
* R. Sedgewick. Implementing quicksort programs, Comm. ACM, 21(10):847-857, 1978.
* David Musser. Introspective Sorting and Selection Algorithms, Software Practice and Experience vol 27, number 8, pages 983-993, 1997
* [Donald Knuth](http://hu.wikipedia.org/wiki/Donald_Knuth). *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. [ISBN 0-201-89685-0](http://hu.wikipedia.org/wiki/Speci%C3%A1lis:K%C3%B6nyvforr%C3%A1sok/0201896850). Pages 113–122 of section 5.2.2: Sorting by Exchanging.
* [Thomas H. Cormen](http://hu.wikipedia.org/w/index.php?title=Thomas_H._Cormen&action=edit&redlink=1), [Charles E. Leiserson](http://hu.wikipedia.org/w/index.php?title=Charles_E._Leiserson&action=edit&redlink=1), [Ronald L. Rivest](http://hu.wikipedia.org/w/index.php?title=Ronald_L._Rivest&action=edit&redlink=1), and [Clifford Stein](http://hu.wikipedia.org/w/index.php?title=Clifford_Stein&action=edit&redlink=1). *[Introduction to Algorithms](http://hu.wikipedia.org/w/index.php?title=Introduction_to_Algorithms&action=edit&redlink=1" \o "Introduction to Algorithms (a lap nem létezik))*, Second Edition. [MIT Press](http://hu.wikipedia.org/w/index.php?title=MIT_Press&action=edit&redlink=1" \o "MIT Press (a lap nem létezik)) and [McGraw-Hill](http://hu.wikipedia.org/wiki/McGraw-Hill), 2001. [ISBN 0-262-03293-7](http://hu.wikipedia.org/wiki/Speci%C3%A1lis:K%C3%B6nyvforr%C3%A1sok/0262032937). Chapter 7: Quicksort, pp. 145–164.
* A. LaMarca and R. E. Ladner. "The Influence of Caches on the Performance of Sorting." Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 1997. pp. 370–379.
* [Faron Moller](http://hu.wikipedia.org/w/index.php?title=Faron_Moller&action=edit&redlink=1). [Analysis of Quicksort](http://www.cs.swan.ac.uk/~csfm/Courses/CS_332/quicksort.pdf). CS 332: Designing Algorithms. Department of Computer Science, University of Wales Swansea.
* Steven Skiena. [Lecture 5 - quicksort](http://www.cs.sunysb.edu/~algorith/lectures-good/node5.html). CSE 373/548 - Analysis of Algorithms. Department of Computer Science. [State University of New York at Stony Brook](http://hu.wikipedia.org/w/index.php?title=State_University_of_New_York_at_Stony_Brook&action=edit&redlink=1" \o "State University of New York at Stony Brook (a lap nem létezik)).
* Conrado Martínez and Salvador Roura, *Optimal sampling strategies in quicksort and quickselect.* SIAM J. Computing 31(3):683-705, 2001.
* Jon L. Bentley and M. Douglas McIlroy, "Engineering a Sort Function, SOFTWARE---PRACTICE AND EXPERIENCE, VOL. 23(11), 1249—1265, 1993

Hivatkozások [[szerkesztés](http://hu.wikipedia.org/w/index.php?title=Gyorsrendez%C3%A9s&action=edit&section=8)]

1. [↑](http://hu.wikipedia.org/wiki/Gyorsrendez%C3%A9s#cite_ref-1) [*Timeline of Computer History: 1960*](http://www.computerhistory.org/timeline/?year=1960). Computer History Museum

További információk [[szerkesztés](http://hu.wikipedia.org/w/index.php?title=Gyorsrendez%C3%A9s&action=edit&section=9)]

* [Gyorsrendezés egy online Javascript IDE-ben](http://tide4javascript.com/?s=Quicksort)
* [Gyorsrendezés különböző programnyelveken](http://en.literateprograms.org/Category:Quicksort) a LiteratePrograms oldalon

Lásd még [[szerkesztés](http://hu.wikipedia.org/w/index.php?title=Gyorsrendez%C3%A9s&action=edit&section=10)]

Need help with a programming assignment? Get affordable [programming homework help](http://helpwithahomework.com/).

**Quicksort**

Quicksort is a fast sorting algorithm, which is used not only for educational purposes, but widely applied in practice. On the average, it has O(n log n) complexity, making quicksort suitable for sorting big data volumes. The idea of the algorithm is quite simple and once you realize it, you can write quicksort as fast as [bubble sort](http://www.algolist.net/Algorithms/Sorting/Bubble_sort).

**Algorithm**

The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:

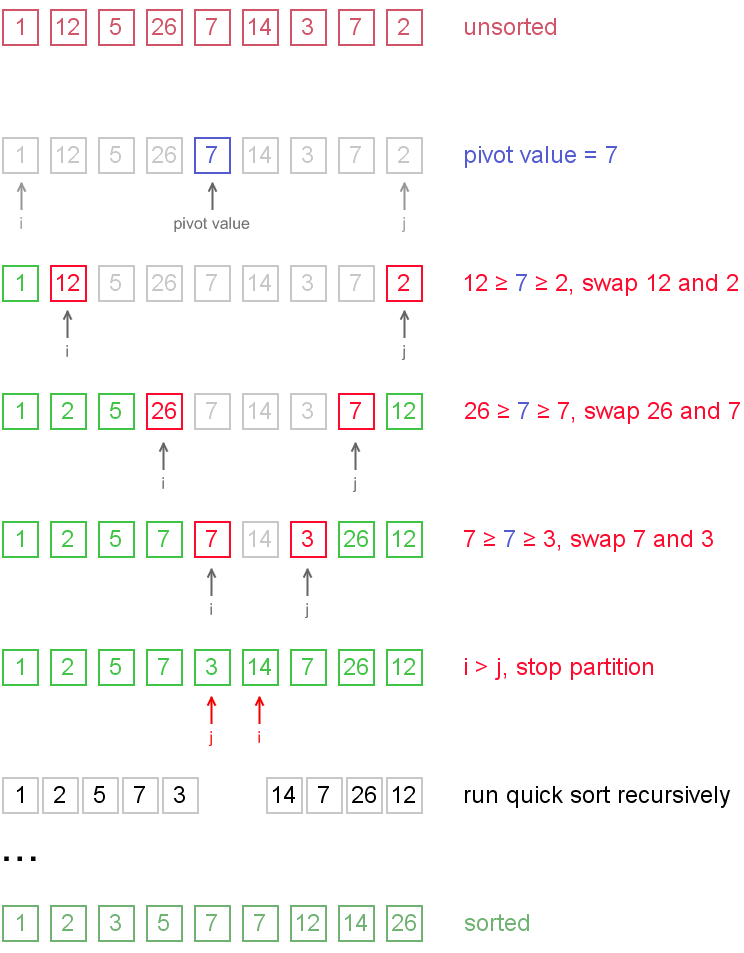
1. **Choose a pivot value.**We take the value of the middle element as pivot value, but it can be any value, which is in range of sorted values, even if it doesn't present in the array.
2. **Partition.**Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.
3. **Sort both parts.**Apply quicksort algorithm recursively to the left and the right parts.

**Partition algorithm in detail**

There are two indices **i** and **j** and at the very beginning of the partition algorithm **i** points to the first element in the array and**j** points to the last one. Then algorithm moves **i** forward, until an element with value greater or equal to the pivot is found. Index **j** is moved backward, until an element with value lesser or equal to the pivot is found. If **i ≤ j**then they are swapped and i steps to the next position (**i + 1**), j steps to the previous one **(j - 1)**. Algorithm stops, when**i** becomes greater than **j**.

After partition, all values before **i-th** element are less or equal than the pivot and all values after **j-th** element are greater or equal to the pivot.

*Example.*Sort {1, 12, 5, 26, 7, 14, 3, 7, 2} using quicksort.



Notice, that we show here only the first recursion step, in order not to make example too long. But, in fact, {1, 2, 5, 7, 3} and {14, 7, 26, 12} are sorted then recursively.

**Why does it work?**

On the partition step algorithm divides the array into two parts and every element **a** from the left part is less or equal than every element **b** from the right part. Also **a** and **b** satisfy **a ≤ pivot ≤ b** inequality. After completion of the recursion calls both of the parts become sorted and, taking into account arguments stated above, the whole array is sorted.

**Complexity analysis**

On the average quicksort has O(n log n) complexity, but strong proof of this fact is not trivial and not presented here. Still, you can find the proof in [[1]](http://www.algolist.net/Algorithms/Sorting/Quicksort#cormen_book). In worst case, quicksort runs O(n2) time, but on the most "practical" data it works just fine and outperforms other O(n log n) sorting algorithms.

**Code snippets**

Partition algorithm is important per se, therefore it may be carried out as a separate function. The code for C++ contains solid function for quicksort, but Java code contains two separate functions for partition and sort, accordingly.

**Java**

**int** partition(**int** arr[], **int** left, **int** right)

{

**int** i = left, j = right;

**int** tmp;

**int** pivot = arr[(left + right) / 2];

**while** (i <= j) {

**while** (arr[i] < pivot)

                  i++;

**while** (arr[j] > pivot)

                  j--;

**if** (i <= j) {

                  tmp = arr[i];

                  arr[i] = arr[j];

                  arr[j] = tmp;

                  i++;

                  j--;

            }

      };

**return** i;

}

**void** quickSort(**int** arr[], **int** left, **int** right) {

**int** index = partition(arr, left, right);

**if** (left < index - 1)

            quickSort(arr, left, index - 1);

**if** (index < right)

            quickSort(arr, index, right);

}

**C++**

void quickSort(int arr[], int left, int right) {

      int i = left, j = right;

      int tmp;

      int pivot = arr[(left + right) / 2];

      /\* partition \*/

      while (i <= j) {

            while (arr[i] < pivot)

                  i++;

            while (arr[j] > pivot)

                  j--;

            if (i <= j) {

                  tmp = arr[i];

                  arr[i] = arr[j];

                  arr[j] = tmp;

                  i++;

                  j--;

            }

      };

      /\* recursion \*/

      if (left < j)

            quickSort(arr, left, j);

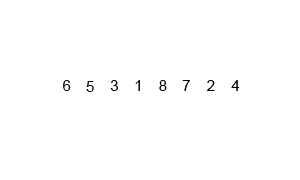
      if (i < right)

            quickSort(arr, i, right);

}

<http://www.zentut.com/c-tutorial/c-merge-sort/>

Introduction to the merge sort algorithm

The merge sort works based on divide and conquer strategy as follows:

1. First, we divide the unsorted list into n sub-lists  Each sub-list contain 1 element, a list with 1 element is considered ordered or sorted.
2. Second, we merge sub-lists repeatedly to make new sub-lists until there is only one sub-list remaining. This sub-list is the final sorted list.

The complexity of the merge sort algorithm is O(NlogN) where N is the number of elements to sort.

For example, to sort a list of integers 5,6,3,1,7,8,2,4 we do it as illustrated in picture.

C merge sort implementation

The following program demonstrate how to implement the merge sort in C. Notice the [recursion technique](http://www.zentut.com/c-tutorial/c-recursive-function/) is used.

[?](http://www.zentut.com/c-tutorial/c-merge-sort/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96 | #include <stdio.h>  #include <stdlib.h>    #define SIZE 8    void merge(int a[], int tmp[], int left, int mid, int right);  void msort(int a[], int tmp[], int left, int right);  void merge\_sort(int a[], int tmp[], const int size);  void display(int a[],const int size);    int main()  {      int a[SIZE] = {5,6,3,1,7,8,2,4};      int tmp[SIZE];        printf("--- C Merge Sort Demonstration --- \n");        printf("Array before sorting:\n");      display(a,SIZE);        merge\_sort(a, tmp, SIZE);        printf("Array after sorting:\n");      display(a,SIZE);        return 0;  }    void merge\_sort(int a[], int tmp[], const int size)  {      msort(a, tmp, 0, size - 1);  }    void msort(int a[], int tmp[], int left, int right)  {      int mid;      if (right > left)      {          mid = (right + left) / 2;          msort(a, tmp, left, mid);          msort(a, tmp, mid + 1, right);          merge(a, tmp, left, mid + 1, right);      }  }    void merge(int a[], int tmp[], int left, int mid, int right)  {      int i, left\_end, count, tmp\_pos;      left\_end = mid - 1;      tmp\_pos = left;      count = right - left + 1;        while ((left <= left\_end) && (mid <= right))      {          if (a[left] <= a[mid])          {              tmp[tmp\_pos] = a[left];              tmp\_pos = tmp\_pos + 1;              left = left +1;          }          else          {              tmp[tmp\_pos] = a[mid];              tmp\_pos = tmp\_pos + 1;              mid = mid + 1;          }      }        while (left <= left\_end)      {          tmp[tmp\_pos] = a[left];          left = left + 1;          tmp\_pos = tmp\_pos + 1;      }      while (mid <= right)      {          tmp[tmp\_pos] = a[mid];          mid = mid + 1;          tmp\_pos = tmp\_pos + 1;      }        for (i = 0; i <= count; i++)      {          a[right] = tmp[right];          right = right - 1;      }  }    void display(int a[],const int size)  {      int i;      for(i = 0; i < size; i++)          printf("%d ",a[i]);        printf("\n");  } |

The output of the program is as follows:

[?](http://www.zentut.com/c-tutorial/c-merge-sort/)

|  |  |
| --- | --- |
| 1  2  3  4  5 | --- C Merge Sort Demonstration ---  Array before sorting:  5 6 3 1 7 8 2 4  Array after sorting:  1 2 3 4 5 6 7 8 |

In this tutorial, we have introduced you the merge sort algorithm and shown you how to implement it in C.

Related Tutorials

* [Selection Sort in C](http://www.zentut.com/c-tutorial/c-selection-sort/)
* [C Quicksort Algorithm](http://www.zentut.com/c-tutorial/c-quicksort-algorithm/)
* [C Bubble Sort](http://www.zentut.com/c-tutorial/c-bubble-sort/)
* [C Shell Sort](http://www.zentut.com/c-tutorial/c-shell-sort/)
* [C Heapsort](http://www.zentut.com/c-tutorial/c-heapsort/)
* [C Insertion Sort](http://www.zentut.com/c-tutorial/insertion-sort-in-c/)

[Previous Tutorial:C Insertion Sort](http://www.zentut.com/c-tutorial/insertion-sort-in-c/)

[Next Tutorial:](http://www.zentut.com/c-tutorial/c-shell-sort/)

<http://www.youtube.com/watch?v=8hHWpuAPBHo>

<http://www.youtube.com/watch?v=ywWBy6J5gz8>

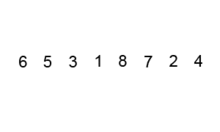
<http://en.wikipedia.org/wiki/Bubble_sort>

Buborékrendezés

Ez a lap egy ellenőrzött változatarészletek megjelenítése/elrejtése

 [[bevezető szerkesztése](http://hu.wikipedia.org/w/index.php?title=Bubor%C3%A9krendez%C3%A9s&action=edit&section=0&editintro=MediaWiki:Editintro-section-0)]

A Wikipédiából, a szabad enciklopédiából

[](http://hu.wikipedia.org/wiki/F%C3%A1jl:Bubble-sort-example-300px.gif)

[http://bits.wikimedia.org/static-1.22wmf1/skins/common/images/magnify-clip.png](http://hu.wikipedia.org/wiki/F%C3%A1jl:Bubble-sort-example-300px.gif)

Buborékrendezésre egy példa

A **buborékrendezés** (angolul: **Bubble sort**) egy egyszerű [algoritmus](http://hu.wikipedia.org/wiki/Algoritmus), amellyel egy véges (nem feltétlenül numerikus) [sorozat](http://hu.wikipedia.org/wiki/Sorozat_(matematika)) – vagy számítástechnikai szóhasználattal élve egy [tömb](http://hu.wikipedia.org/wiki/T%C3%B6mb_(adatszerkezet)) – elemei sorba rendezhetők [(n-1)n]/2 összehasonlítás elvégzésével, ahol n a [sorozat](http://hu.wikipedia.org/wiki/Sorozat_(matematika))elemeinek számát jelenti.

Mivel az algoritmus nem túl hatékony, a gyakorlatban szinte egyáltalán nem, inkább csak az algoritmuselmélet oktatása során használják.

|  |
| --- |
| **Tartalomjegyzék**    [[elrejtés](http://hu.wikipedia.org/wiki/Bubor%C3%A9krendez%C3%A9s)]   * [1 Az algoritmus](http://hu.wikipedia.org/wiki/Bubor%C3%A9krendez%C3%A9s#Az_algoritmus) * [2 Lásd még](http://hu.wikipedia.org/wiki/Bubor%C3%A9krendez%C3%A9s#L.C3.A1sd_m.C3.A9g) * [3 Források](http://hu.wikipedia.org/wiki/Bubor%C3%A9krendez%C3%A9s#Forr.C3.A1sok) * [4 További információk](http://hu.wikipedia.org/wiki/Bubor%C3%A9krendez%C3%A9s#Tov.C3.A1bbi_inform.C3.A1ci.C3.B3k) |

Az algoritmus [[szerkesztés](http://hu.wikipedia.org/w/index.php?title=Bubor%C3%A9krendez%C3%A9s&action=edit&section=1)]

Az algoritmus alkalmazásának feltétele hogy a [sorozat](http://hu.wikipedia.org/wiki/Sorozat_(matematika)) elemeihez létezzen egy rendezési [reláció](http://hu.wikipedia.org/wiki/Rel%C3%A1ci%C3%B3).

Az algoritmus két egymásba ágyazott ciklusból áll. A tömb első elemének indexe az 1, elemeinek száma pedig n. Az elemek itt számok, és a reláció a > (nagyobb).

**CIKLUS** i = n **TŐL** 2 **IG** {

**CIKLUS** j = 1 **TŐL** i-1 **IG** {

**HA** TOMB[j] > TOMB[j+1] **AKKOR** {

**CSERÉLD FEL ŐKET:** TOMB[j], TOMB[j+1]

}

}

}

A futás befejezése után a tömb 1-es indexű eleme lesz a legkisebb és az n indexű a legnagyobb.

Az algoritmus onnan kapta a nevét, hogy először a legnagyobb elem "száll fel" a tömb utolsó helyére, utána a második legnagyobb az azt követő helyre, és így tovább, mint ahogy a buborékok szállnak felfelé egy pohárban.

Lásd még [[szerkesztés](http://hu.wikipedia.org/w/index.php?title=Bubor%C3%A9krendez%C3%A9s&action=edit&section=2)]

* [Koktélrendezés](http://hu.wikipedia.org/wiki/Kokt%C3%A9lrendez%C3%A9s)
* [Fésűs rendezés](http://hu.wikipedia.org/wiki/F%C3%A9s%C5%B1s_rendez%C3%A9s)

**Bubble Sort**

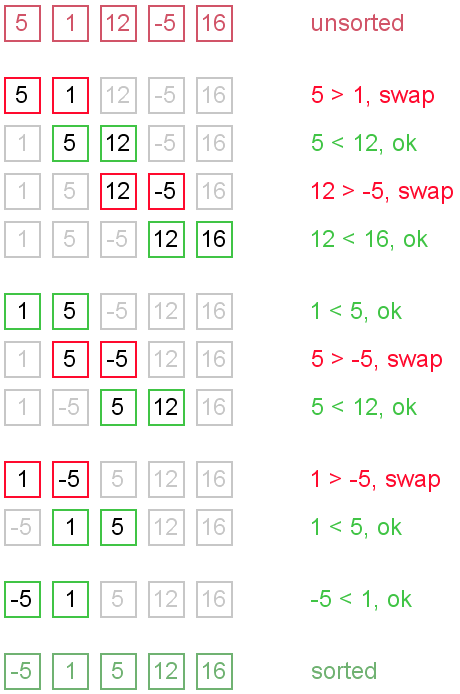
Bubble sort is a simple and well-known sorting algorithm. It is used in practice once in a blue moon and its main application is to make an introduction to the sorting algorithms. Bubble sort belongs to O(n2) sorting algorithms, which makes it quite inefficient for sorting large data volumes. Bubble sort is **stable** and **adaptive**.

**Algorithm**

1. Compare each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them.
2. If at least one swap has been done, repeat step 1.

You can imagine that on every step big bubbles float to the surface and stay there. At the step, when no bubble moves, sorting stops. Let us see an example of sorting an array to make the idea of bubble sort clearer.

*Example.*Sort {5, 1, 12, -5, 16} using bubble sort.



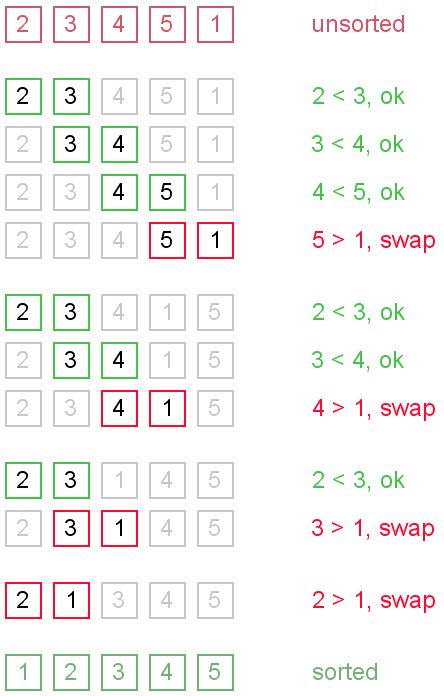
**Complexity analysis**

Average and worst case complexity of bubble sort is O(n2). Also, it makes O(n2) swaps in the worst case. Bubble sort is adaptive. It means that for almost sorted array it gives O(n) estimation. Avoid implementations, which don't check if the array is already sorted on every step (any swaps made). This check is necessary, in order to preserve adaptive property.

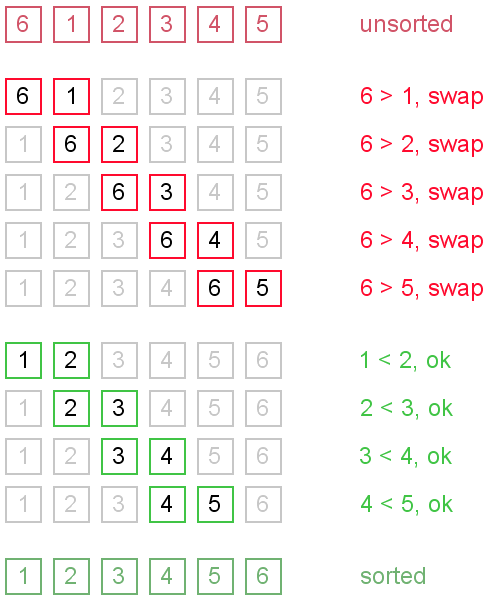
**Turtles and rabbits**

One more problem of bubble sort is that its running time badly depends on the initial order of the elements. Big elements (rabbits) go up fast, while small ones (turtles) go down very slow. This problem is solved in the Cocktail sort.

**Turtle example.**Thought, array {2, 3, 4, 5, 1} is almost sorted, it takes O(n2) iterations to sort an array. Element {1} is a turtle.



**Rabbit example.**Array {6, 1, 2, 3, 4, 5} is almost sorted too, but it takes O(n) iterations to sort it. Element {6} is a rabbit. This example demonstrates adaptive property of the bubble sort.



**Code snippets**

There are several ways to implement the bubble sort. Notice, that "swaps" check is absolutely necessary, in order to preserve adaptive property.

**Java**

**public** **void** bubbleSort(**int**[] arr) {

**boolean** swapped = **true**;

**int** j = 0;

**int** tmp;

**while** (swapped) {

            swapped = **false**;

            j++;

**for** (**int** i = 0; i < arr.length - j; i++) {

**if** (arr[i] > arr[i + 1]) {

                        tmp = arr[i];

                        arr[i] = arr[i + 1];

                        arr[i + 1] = tmp;

                        swapped = **true**;

                  }

            }

      }

}

**C++**

void bubbleSort(int arr[], int n) {

      bool swapped = true;

      int j = 0;

      int tmp;

      while (swapped) {

            swapped = false;

            j++;

            for (int i = 0; i < n - j; i++) {

                  if (arr[i] > arr[i + 1]) {

                        tmp = arr[i];

                        arr[i] = arr[i + 1];

                        arr[i + 1] = tmp;

                        swapped = true;

                  }

            }

      }

}<http://www.cs.oswego.edu/~mohammad/classes/csc241/samples/sort/Sort2-E.html>

<http://www.sorting-algorithms.com/bubble-sort>

Bubble Sort

[http://www.sorting-algorithms.com/static/img/reload-48x48.jpg](javascript:sortRestart('*','*');)Problem Size:  [20](javascript:selectSize(20);) · [30](javascript:selectSize(30);) · [40](javascript:selectSize(40);) · [50](javascript:selectSize(50);)     Magnification:  [1x](javascript:selectResolution(1);) · [2x](javascript:selectResolution(2);) · [3x](javascript:selectResolution(3);)

Algorithm:  [Insertion](http://www.sorting-algorithms.com/insertion-sort) · [Selection](http://www.sorting-algorithms.com/selection-sort) · [Bubble](http://www.sorting-algorithms.com/bubble-sort) · [Shell](http://www.sorting-algorithms.com/shell-sort) · [Merge](http://www.sorting-algorithms.com/merge-sort) · [Heap](http://www.sorting-algorithms.com/heap-sort) · [Quick](http://www.sorting-algorithms.com/quick-sort) · [Quick3](http://www.sorting-algorithms.com/quick-sort-3-way)

|  |  |  |  |
| --- | --- | --- | --- |
| [Random](http://www.sorting-algorithms.com/random-initial-order)  http://www.sorting-algorithms.com/animation/40/random-initial-order/bubble-sort.gif | [Nearly Sorted](http://www.sorting-algorithms.com/nearly-sorted-initial-order)  http://www.sorting-algorithms.com/animation/40/nearly-sorted-initial-order/bubble-sort.gif | [Reversed](http://www.sorting-algorithms.com/reversed-initial-order)  http://www.sorting-algorithms.com/animation/40/reversed-initial-order/bubble-sort.gif | [Few Unique](http://www.sorting-algorithms.com/few-unique-keys)  http://www.sorting-algorithms.com/animation/40/few-unique-keys/bubble-sort.gif |

Algorithm

for i = 1:n,

swapped = false

for j = n:i+1,

if a[j] < a[j-1],

swap a[j,j-1]

swapped = true

*→ invariant: a[1..i] in final position*

break if not swapped

end

Properties

* Stable
* O(1) extra space
* O(n2) comparisons and swaps
* Adaptive: O(n) when nearly sorted

Discussion

Bubble sort has many of the same properties as insertion sort, but has slightly higher overhead. In the case of nearly sorted data, bubble sort takes O(n) time, but requires at least 2 passes through the data (whereas insertion sort requires something more like 1 pass).

Directions

* Click on http://www.sorting-algorithms.com/static/img/reload-24x24.gif above to restart the animations in a row, a column, or the entire table.
* Click directly on an animation image to start or restart it.
* Click on a problem size number to reset all animations.

Key

Bubble sort

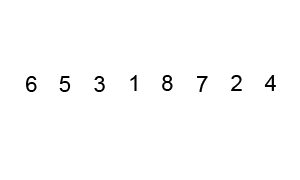
From Wikipedia, the free encyclopedia

|  |  |
| --- | --- |
| **Bubble sort** | |
| [Static visualization of bubblesort](http://en.wikipedia.org/wiki/File:Bubblesort-edited.png) | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n^2) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n^2) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(1) auxiliary |

**Bubble sort**, sometimes incorrectly referred to as **sinking sort**, is a simple [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and [swapping](http://en.wikipedia.org/wiki/Swap_(computer_science)) them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort). Although the algorithm is simple, most of the other sorting algorithms are more efficient for large lists.

|  |
| --- |
| **Contents**    [[hide](http://en.wikipedia.org/wiki/Bubble_sort)]   * [1 Analysis](http://en.wikipedia.org/wiki/Bubble_sort#Analysis)   + [1.1 Performance](http://en.wikipedia.org/wiki/Bubble_sort#Performance)   + [1.2 Rabbits and turtles](http://en.wikipedia.org/wiki/Bubble_sort#Rabbits_and_turtles)   + [1.3 Step-by-step example](http://en.wikipedia.org/wiki/Bubble_sort#Step-by-step_example) * [2 Implementation](http://en.wikipedia.org/wiki/Bubble_sort#Implementation)   + [2.1 Pseudocode implementation](http://en.wikipedia.org/wiki/Bubble_sort#Pseudocode_implementation)   + [2.2 Optimizing bubble sort](http://en.wikipedia.org/wiki/Bubble_sort#Optimizing_bubble_sort) * [3 In practice](http://en.wikipedia.org/wiki/Bubble_sort#In_practice) * [4 Variations](http://en.wikipedia.org/wiki/Bubble_sort#Variations) * [5 Misnomer](http://en.wikipedia.org/wiki/Bubble_sort#Misnomer) * [6 Notes](http://en.wikipedia.org/wiki/Bubble_sort#Notes) * [7 References](http://en.wikipedia.org/wiki/Bubble_sort#References) * [8 External links](http://en.wikipedia.org/wiki/Bubble_sort#External_links) |

[[edit](http://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=1)]Analysis

[](http://en.wikipedia.org/wiki/File:Bubble-sort-example-300px.gif)

[http://bits.wikimedia.org/static-1.22wmf2/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Bubble-sort-example-300px.gif)

An example on bubble sort. Starting from the beginning of the list, compare every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one). After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.

[[edit](http://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=2)]**Performance**

Bubble sort has worst-case and average complexity both [*О*](http://en.wikipedia.org/wiki/Big_o_notation)(*n*2), where *n* is the number of items being sorted. There exist many sorting algorithms with substantially better worst-case or average complexity of *O*(*n* log *n*). Even other *О*(*n*2) sorting algorithms, such as [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort), tend to have better performance than bubble sort. Therefore, bubble sort is not a practical sorting algorithm when *n* is large.

The only significant advantage that bubble sort has over most other implementations, even [quicksort](http://en.wikipedia.org/wiki/Quicksort" \o "Quicksort), but not [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort), is that the ability to detect that the list is sorted is efficiently built into the algorithm. Performance of bubble sort over an already-sorted list (best-case) is *O*(*n*). By contrast, most other algorithms, even those with better [average-case complexity](http://en.wikipedia.org/wiki/Average-case_complexity), perform their entire sorting process on the set and thus are more complex. However, not only does [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) have this mechanism too, but it also performs better on a list that is substantially sorted (having a small number of [inversions](http://en.wikipedia.org/wiki/Inversion_(discrete_mathematics))).

[[edit](http://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=3)]**Rabbits and turtles**

The positions of the elements in bubble sort will play a large part in determining its performance. Large elements at the beginning of the list do not pose a problem, as they are quickly swapped. Small elements towards the end, however, move to the beginning extremely slowly. This has led to these types of elements being named rabbits and turtles, respectively.

Various efforts have been made to eliminate turtles to improve upon the speed of bubble sort. [Cocktail sort](http://en.wikipedia.org/wiki/Cocktail_sort) is a bi-directional bubble sort that goes from beginning to end, and then reverses itself, going end to beginning. It can move turtles fairly well, but it retains [*O(n2)*](http://en.wikipedia.org/wiki/Big_O_notation) worst-case complexity. [Comb sort](http://en.wikipedia.org/wiki/Comb_sort) compares elements separated by large gaps, and can move turtles extremely quickly before proceeding to smaller and smaller gaps to smooth out the list. Its average speed is comparable to faster algorithms like [quicksort](http://en.wikipedia.org/wiki/Quicksort" \o "Quicksort).

[[edit](http://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=4)]**Step-by-step example**

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

**First Pass:**  
( **5** **1** 4 2 8 ) \to ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.  
( 1 **5** **4** 2 8 ) \to ( 1 **4** **5** 2 8 ), Swap since 5 > 4  
( 1 4 **5** **2** 8 ) \to ( 1 4 **2** **5** 8 ), Swap since 5 > 2  
( 1 4 2 **5** **8** ) \to ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.  
**Second Pass:**  
( **1** **4** 2 5 8 ) \to ( **1** **4** 2 5 8 )  
( 1 **4** **2** 5 8 ) \to ( 1 **2** **4** 5 8 ), Swap since 4 > 2  
( 1 2 **4** **5** 8 ) \to ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) \to ( 1 2 4 **5** **8** )  
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.  
**Third Pass:**  
( **1** **2** 4 5 8 ) \to ( **1** **2** 4 5 8 )  
( 1 **2** **4** 5 8 ) \to ( 1 **2** **4** 5 8 )  
( 1 2 **4** **5** 8 ) \to ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) \to ( 1 2 4 **5** **8** )

[[edit](http://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=5)]Implementation

[[edit](http://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=6)]**Pseudocode implementation**

The algorithm can be expressed as (0-based array):

procedure bubbleSort( A : list of sortable items )

repeat

swapped = false

for i = 1 to length(A) - 1 inclusive do:

*/\* if this pair is out of order \*/*

if A[i-1] > A[i] then

*/\* swap them and remember something changed \*/*

swap( A[i-1], A[i] )

swapped = true

end if

end for

until not swapped

end procedure

To reiterate, there are better sorting algorithms like [Quicksort](http://en.wikipedia.org/wiki/Quicksort" \o "Quicksort) to use in most practical applications, and most programming languages also include a built-in sort function in the standard library which should be used instead of implementing ones own sorting algorithm.

[[edit](http://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=7)]**Optimizing bubble sort**

The bubble sort algorithm can be easily optimized by observing that the n-th pass finds the n-th largest element and puts it into its final place. So, the inner loop can avoid looking at the last n-1 items when running for the n-th time:

procedure bubbleSort( A : list of sortable items )

n = length(A)

repeat

swapped = false

for i = 1 to n-1 inclusive do

if A[i-1] > A[i] then

swap(A[i-1], A[i])

swapped = true

end if

end for

n = n - 1

until not swapped

end procedure

More generally, it can happen that more than one element is placed in their final position on a single pass. In particular, after every pass, all elements after the last swap are sorted, and do not need to be checked again. This allows us to skip over a lot of the elements, resulting in about a worst case 50% improvement in comparison count (though no improvement in swap counts), and adds very little complexity because the new code subsumes the "swapped" variable:

To accomplish this in [pseudocode](http://en.wikipedia.org/wiki/Pseudocode" \o "Pseudocode) we write the following:

procedure bubbleSort( A : list of sortable items )

n = length(A)

repeat

newn = 0

for i = 1 to n-1 inclusive do

if A[i-1] > A[i] then

swap(A[i-1], A[i])

newn = i

end if

end for

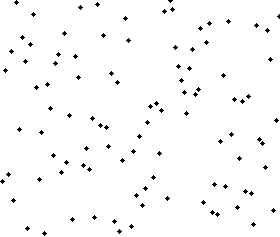
n = newn

until n = 0

end procedure

Alternate modifications, such as the [cocktail shaker sort](http://en.wikipedia.org/wiki/Cocktail_shaker_sort) attempt to improve on the bubble sort performance while keeping the same idea of repeatedly comparing and swapping adjacent items.

[[edit](http://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=8)]In practice

[](http://en.wikipedia.org/wiki/File:Bubble_sort_animation.gif)

[http://bits.wikimedia.org/static-1.22wmf2/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Bubble_sort_animation.gif)

A bubble sort, a sorting algorithm that continuously steps through a list, [swapping](http://en.wikipedia.org/wiki/Swap_(computer_science)) items until they appear in the correct order. Note that the largest end gets sorted first, with smaller elements taking longer to move to their correct positions.

Although bubble sort is one of the simplest sorting algorithms to understand and implement, its [*O(n2)*](http://en.wikipedia.org/wiki/Big_O_notation) complexity means that its efficiency decreases dramatically on lists of more than a small number of elements. Even among simple *O(n2)* sorting algorithms, algorithms like [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) are usually considerably more efficient.

Due to its simplicity, bubble sort is often used to introduce the concept of an algorithm, or a sorting algorithm, to introductory[computer science](http://en.wikipedia.org/wiki/Computer_science) students. However, some researchers such as [Owen Astrachan](http://en.wikipedia.org/wiki/Owen_Astrachan) have gone to great lengths to disparage bubble sort and its continued popularity in computer science education, recommending that it no longer even be taught.[[1]](http://en.wikipedia.org/wiki/Bubble_sort#cite_note-Astrachan2003-1)

The [Jargon file](http://en.wikipedia.org/wiki/Jargon_file), which famously calls [bogosort](http://en.wikipedia.org/wiki/Bogosort" \o "Bogosort) "the archetypical [sic] perversely awful algorithm", also calls bubble sort "the generic **bad** algorithm".[[2]](http://en.wikipedia.org/wiki/Bubble_sort#cite_note-2) [Donald Knuth](http://en.wikipedia.org/wiki/Donald_Knuth), in his famous book [*The Art of Computer Programming*](http://en.wikipedia.org/wiki/The_Art_of_Computer_Programming), concluded that "the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems", some of which he then discusses.[[3]](http://en.wikipedia.org/wiki/Bubble_sort#cite_note-Knuth-3)

Bubble sort is [asymptotically](http://en.wikipedia.org/wiki/Asymptotic_notation) equivalent in running time to [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) in the worst case, but the two algorithms differ greatly in the number of swaps necessary. Experimental results such as those of Astrachan have also shown that [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort)performs considerably better even on random lists. For these reasons many modern algorithm textbooks avoid using the bubble sort algorithm in favor of insertion sort.

Bubble sort also interacts poorly with modern CPU hardware. It requires at least twice as many writes as insertion sort, twice as many cache misses, and asymptotically more [branch mispredictions](http://en.wikipedia.org/wiki/Branch_prediction). Experiments by Astrachan sorting strings in Java show bubble sort to be roughly 5 times slower than [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) and 40% slower than [selection sort](http://en.wikipedia.org/wiki/Selection_sort).[[1]](http://en.wikipedia.org/wiki/Bubble_sort#cite_note-Astrachan2003-1)

In computer graphics it is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n). For example, it is used in a polygon filling algorithm, where bounding lines are sorted by their x coordinate at a specific scan line (a line parallel to x axis) and with incrementing y their order changes (two elements are swapped) only at intersections of two lines.

[[edit](http://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=9)]Variations

* [Odd-even sort](http://en.wikipedia.org/wiki/Odd-even_sort) is a parallel version of bubble sort, for message passing systems.
* [Cocktail sort](http://en.wikipedia.org/wiki/Cocktail_sort) is another parallel version of the bubble sort
* In some cases, the sort works from right to left (the opposite direction), which is more appropriate for partially sorted lists, or lists with unsorted items added to the end.

[[edit](http://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=10)]Misnomer

Bubble sort has incorrectly been called sinking sort. Sinking sort is correctly an alias for insertion sort. This error is largely due to the National Institute of Standards and Technology listing sinking sort as an alias for bubble sort. [[1]](http://xlinux.nist.gov/dads/HTML/bubblesort.html) In Donald Knuth's *The Art of Computer Programming*, Volume 3: *Sorting and Searching* he states in section 5.2.1 'Sorting by Insertion', that [the value] "settles to its proper level" this method of sorting has often been called the *sifting* or *sinking* technique. Furthermore the *larger* values might be regarded as *heavier* and therefore be seen to progressively *sink* to the *bottom* of the list, leading to the misnomer.

To clarify, we can also observe the behavior of the two algorithms. In bubble sort, the larger bubbles (higher values) bubble up displacing the smaller bubbles (lower values). Insertion on the other hand, sinks each successive value down to its correct location in the sorted portion of the collection.

[[edit](http://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=11)]Notes

1. ^ [***a***](http://en.wikipedia.org/wiki/Bubble_sort#cite_ref-Astrachan2003_1-0) [***b***](http://en.wikipedia.org/wiki/Bubble_sort#cite_ref-Astrachan2003_1-1) Owen Astrachan. Bubble Sort: An Archaeological Algorithmic Analysis. SIGCSE 2003 Hannan Akhtar . [(pdf)](http://www.cs.duke.edu/~ola/papers/bubble.pdf)
2. [**^**](http://en.wikipedia.org/wiki/Bubble_sort#cite_ref-2) <http://www.jargon.net/jargonfile/b/bogo-sort.html>
3. [**^**](http://en.wikipedia.org/wiki/Bubble_sort#cite_ref-Knuth_3-0) [Donald Knuth](http://en.wikipedia.org/wiki/Donald_Knuth). [*The Art of Computer Programming*](http://en.wikipedia.org/wiki/The_Art_of_Computer_Programming), Volume 3: *Sorting and Searching*, Second Edition. Addison-Wesley, 1998. [ISBN 0-201-89685-0](http://en.wikipedia.org/wiki/Special:BookSources/0201896850). Pages 106–110 of section 5.2.2: Sorting by Exchanging.

[[edit](http://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=12)]References

* [Donald Knuth](http://en.wikipedia.org/wiki/Donald_Knuth). *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. [ISBN 0-201-89685-0](http://en.wikipedia.org/wiki/Special:BookSources/0201896850). Pages 106–110 of section 5.2.2: Sorting by Exchanging.
* [Thomas H. Cormen](http://en.wikipedia.org/wiki/Thomas_H._Cormen), [Charles E. Leiserson](http://en.wikipedia.org/wiki/Charles_E._Leiserson), [Ronald L. Rivest](http://en.wikipedia.org/wiki/Ronald_L._Rivest), and [Clifford Stein](http://en.wikipedia.org/wiki/Clifford_Stein). [*Introduction to Algorithms*](http://en.wikipedia.org/wiki/Introduction_to_Algorithms), Second Edition. MIT Press and McGraw-Hill, 2001. [ISBN 0-262-03293-7](http://en.wikipedia.org/wiki/Special:BookSources/0262032937). Problem 2-2, pg.38.
* [Sorting in the Presence of Branch Prediction and Caches](https://www.cs.tcd.ie/publications/tech-reports/reports.05/TCD-CS-2005-57.pdf)
* Fundamentals of Data Structures by Ellis Horowitz, [Sartaj Sahni](http://en.wikipedia.org/wiki/Sartaj_Sahni" \o "Sartaj Sahni) and Susan Anderson-Freed [ISBN 81-7371-605-6](http://en.wikipedia.org/wiki/Special:BookSources/8173716056)

[[edit](http://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=13)]External links

|  |  |
| --- | --- |
| http://upload.wikimedia.org/wikipedia/commons/thumb/d/df/Wikibooks-logo-en-noslogan.svg/40px-Wikibooks-logo-en-noslogan.svg.png | The Wikibook [*Algorithm implementation*](http://en.wikibooks.org/wiki/Algorithm_implementation) has a page on the topic of: [***Bubble sort***](http://en.wikibooks.org/wiki/Algorithm_implementation/Sorting/Bubble_sort) |

|  |  |
| --- | --- |
| http://upload.wikimedia.org/wikipedia/en/thumb/4/4a/Commons-logo.svg/30px-Commons-logo.svg.png | Wikimedia Commons has media related to: [***Bubble sort***](http://commons.wikimedia.org/wiki/Category:Bubble_sort) |

|  |  |
| --- | --- |
| http://upload.wikimedia.org/wikipedia/commons/thumb/9/91/Wikiversity-logo.svg/40px-Wikiversity-logo.svg.png | Wikiversity has learning materials about [***Bubble sort***](http://en.wikiversity.org/wiki/Bubble_sort) |

* [Bubble Sort implemented in 34 languages](http://codecodex.com/wiki/Bubble_sort)
* [Animated Sorting Algorithms: Bubble Sort](http://www.sorting-algorithms.com/bubble-sort) – graphical demonstration and discussion of bubble sort
* [Lafore's Bubble Sort](http://lecture.ecc.u-tokyo.ac.jp/~ueda/JavaApplet/BubbleSort.html) (Java applet animation)

<http://prog.ide.sk/pas2.php?s=34>

**Creating a Turbo Pascal Unit**

There are 3 stages that you must go through, in order to create and use your own library unit.

* Firstly you have to create a .PAS file containing all the program code for your library.
* Secondly you must compile the .PAS file to create a .TPU machine code version of your library.
* Finally, you have to include your library's name in the uses list of your programs.

**What a unit looks like**

You will need to create a .PAS file similar to the one below. It is similar to an ordinary Turbo Pascal program, however there are differences. Note especially the *interface* and *implementation* sections.

unit MyUnit;

interface

uses crt,...etc;

procedure StarGow(x,y: integer; s: string); {Procedure and function headers}

procedure StarLine(y: integer);

function StopsToSpaces(s: string): string;

\*

\*

\* etc.

implementation

{--------------------------------------------------------------------------------------}

procedure StarGow(x,y: integer; s: string); {Procedure and function coding}

begin

gotoxy(x,y);

clreol;

gotoxy(x,y);

write('\*',s,'\*');

end;

{---------------------------------------------------------------------------------------}

procedure StarLine(y: integer);

\*

\*

\*

\* ...etc.

end; {End of last procedure or function}

{---------------------------------------------------------------------------------------}

end. {End of unit}

**To compile the unit**

Before you compile it, you must first choose:

**Compile / Destination**

This will change the destination to disc (click on Compile on the menu bar to check that this has happened). If you then choose **Compile / Compile**, the .TPU file will be created. This is the machine code version of your library.

**To use the unit**

Include the unit's name in the *uses* line at the top of any program that wants access to your library, e.g.

uses crt, dos, MyUnit;

|  |
| --- |
| **Unit Creation** |
|  |

|  |
| --- |
| When writing a program, the main reason for using routines is to isolate assignments. This allows you to effectively troubleshoot problems when they arise. When using routines in a program, we saw that the order of declaring them was important. For example, you cannot call a routine that has not been declared yet. For this reason, whenever you need to call it, you should find out where it was created or whether it has been declared already. If the program is using many routines, it would become cumbersome to start looking for them. At the same time, on a large program, it is usual for many routines to use the same kind of variable. To make these routines easily manageable, you can create a source file where you would list them. Such a file is called a unit and it has the pas extension.  A unit file has the following structure: |

unit Unit1;

interface

implementation

end.

|  |
| --- |
| The source file starts with the unit keyword. This keyword lets the compiler know that this source file is a unit. The unit keyword must be followed by a name for the file. When you add a new unit to your project, Delphi gives it a default name. If it is the first unit, it would be called unit1, the second unit would be called unit2, etc. If you want to change the name of the unit, you must save it. This would prompt you to specify a name for the unit. When naming a unit, you must use a unique unit name in the project: two units must not have the same name in the same project. The name of the unit is followed by a semi-colon.  Under the unit line, a section with the interface keyword starts. In this section, you can declare variables, constants, enumerators, types, procedures, functions, and other objects such as classes we will study eventually. Although the routines are declared for later implementation, you do not have to type the forward keyword on their declaration. Their presence in the interface section indicates already that they will be implemented later on.   At the end of the interface section, you start a new section with the implementation keyword. In this section, you can implement the routines declared in the interface section. You can also use the variables, constants, and other objects declared in the previous section.  Here is an example of a simple unit file: |

unit Unit1;

interface

Function AddTwoNumbers : Double; forward;

function GetNumber : Double; forward;

implementation

Function AddTwoNumbers : Double;

var

Number1 : Double;

Number2 : Double;

begin

Number1 := GetNumber;

Number2 := GetNumber;

Result := Number1 + Number2;

end;

function GetNumber : Double;

var Nbr : Double;

begin

Write('Enter Number: ');

Readln(Nbr);

Result := Nbr;

end;

end.

|  |
| --- |
| **Using a Unit** |
|  |

|  |
| --- |
| Once an Object Pascal unit exists, whether you created it or it shipped with the compiler, you can use its variables, constants, routines or types. To use the contents of a unit, from the file where you want it, you must include it. This is done using the uses keyword.  If a unit is part of the project you are working on, to use it in a certain file, type the uses keyword followed by the name of the unit. If you are want to call more than one unit, include each, separating them by a comma. The list of units must end with a semi-colon. An example would be:  uses Unit1, Unit2, Unit\_etc;  If the unit shipped with the compiler, you can also include just its name. The compiler would know where to find it. If the unit exists in another project, type the uses keyword, followed by the name of the unit. Then type the in keyword followed by the path to the file where the unit is located, in single-quotes. An example would be: |

uses

CustomUnit in ‘C:\My Programs\Customers Orders\AprilOrders.pas’;